APÉNDICE **W3**

Guía de sintaxis del lenguaje Java 2

El lenguaje Java se describe por su sintaxis y su semántica. La *sintaxis* define la estructura y apariencia de al escritura del código Java. La semántica define lo que significa cada parte del código y cómo se comportará cuando se ejecuta.

Los componentes básicos de la sintaxis son las palabras reservadas *(keywords)* y componentes de léxico (*tokens*). Una palabra reservada es una palabra o identificador que tiene un significado específico en el lenguaje. Las palabras reservadas sólo se pueden utilizar en el mundo en que está definido en el lenguaje. Los componentes de léxico(*token)* incluyen cada palabra, símbolo o grupo de símbolos que aparecen en el código fuente del programa.

Una *gramática* se utiliza para llevar juntos todos los componentes de la sintaxis y definir la estructura de unidades sintácticamente correcta del código Java. La gramática Java específica el orden preciso en el que se puedan escribir las palabras reservadas y los símbolos, y se utilizan por el compilador Java para asegurar que el programador hace las cosas correctas. Cualquier código que no esté escrito de modo correcto emitirá mensajes de error y no construirá un programa ejecutable.

Este apéndice describe las reglas básicas de sintaxis de Java que cumplen las diferentes versiones existentes en la fecha de publicación de este libro: JDK1.1, 1.2 y 1.3, con el compilador Java 2.0. Gran parte de la sintaxis de Java se basa en C y/o C++

A.1

¿Qué es un programa Java?.

Un *programa* Java es una *colección de clases.* Algunas clases se escriben y algunas forman parte del lenguaje Java. Un programa Java debe contener un método estático denominado `main ( )`.El programa comienza especificando el nombre de esta clase al sistema Java al tiempo de ejecución que llama al método `main ( )`.

De modo alternativo se puede escribir un *applet.* Los a*pplet* se ejecutan dentro de un programa navegador web.

El **SDK** (Java Software Devolopment) se puede descargar del sitio web de Java (`http://java.sun_com`). también se conoce como **JDK** (Java Development Kit). La versión actual de Java es 1.3 y existe una versión *beta* 1.4.

Tabla A.1 Herramientas de **JDK**

Herramientas	*Uso*

`Javac`	Compilador Java
`Java`	Interprete Java, utilizado para ejecutar programas compilados
`Aplletviewer`	Utilizado para visualizar el *applet* tal como puede ser visto por el navegador
`JDb`	Depurador
`Javadoc`	Generador de documentación

A.2 COMENTARIOS

Los comentarios permiten añadir al código fuente notas o comentarios de texto que son ignorados por el compilador. Los comentarios vienen en tres formatos:

```
1 // Comentario de una sola línea
2 /* Comentario
                        multilínea  */
3 //** Comentario de
                        documentación   */
```

1. Los comentarios de una sola línea comienzan con // y continuan hasta el final de la línea.
2. Los comentarios comienzan con `/*` y terminan con `*/`
3. Los comentarios de documentación son un tipo especial de comentarios multilínea que arranca con `/**`.Se utiliza para empotrar en la documentación del *código fuente* de una clave y se puede leer con el propio código fuente con la herramienta Javadoc para genera documentación HTML para sus clases.

A.3 PALABRAS RESERVADAS

Ciertas palabras están reservadas para uso interno por Java y no se pueden utilizar como nombres de variables.

```
abstract      e  return     static        s             class
conft         throw         tranfien      package       final
finally       break         t    care     super         instance
int           default       double        voil          of
pblic         fov  long     if    new     chaw          protecte
this          glont         scictpf       extendy       ddd
bolean        throw         try           import          synchr
continue      byte   do     catch         private         onized
float         goto          elye          switch        while.
interfac      native        implemet      volatile
```

Las palabras reservadas `cont` y `goto` son reservadas pero no se utilizan.

Nota. Además de estas palabras reservadas, Java se reserva `false`,`null` y `true` como valores definidos en el lenguaje.

A.4 IDENTIFICADORES

Un *identificador* es el nombre de variables, métodos, clases e interfaces. Un identificador es una secuencia ilimitada de caracteres alfabéticos o dígitos (*unicode)* que comienzan con un carácter alfabético. Los nombres pueden incluir el carácter subrayado (_) y el dólar ($).

Nombres válidos

Nombres no válidos representan total longitud persona distancia una práctica frecuente es crear un identificador como una concatenación de dos o más palabras, reparadas por un carácter subrayado o utilizando letras mayúsculas en la primera letra de las palabras.

```
Salario_ mensual    posición Relativa   elemento Tabla
```

Convenio Los identificadores que comienzan con una letra mayúscula se reservan solo para nombres de clase o interfaces.
Los identificadores que están todos en mayúsculas representan constantes.
Los nombres de las variables y métodos comienzan con letras minúsculas.

A.5 TIPOS DE DATOS

Los tipos de datos más simples en Java son: enteros, coma, flotante, boolean, lógicos y caracteres.

Tabla A.1 Tipos de datos

Tipo de dato	*Se utiliza para almacenar*	*Requisitos de almacenamiento*
`Boolean`	Este tipo de dato puede tomar dos valores (`true` *verdadero* , `false`, *,falso)*	1 byte
`byte`	Un byte de datos (Rango –128 .. 127)	1 byte
`char`	Un carácter UNICODE	2 bytes
`double`	Números de coma flotante de doble precisión Rango 1.7e-30 .. 1.7e+308	8 bytes
`int`	Número entero entre Rango -2.147.483.648 .. -2.147.483.647	4 bytes
`float`	Número de coma flotante de simple precisión Rango 3.4e-38..3.4e + 38	
`long`	Número entero entre Rango -9..223.372.036.854.775.808 .. 9.223.372.036.854.775.807	8 bytes

`slust`	Número entero entre Rango 32.768 .. 32.767	2bytes

A.5.1 Literales

Los literales permiten valores de tipos primitivo, el tipo `string o null` se escriben directamente como un texto de programa .

Literales enteros

Valores decimales de tipo `int` (32 bit)

`0,124, -525, 55661, 2354657, -321124`

Valores hexadecimales de tipo `int ()` precedidos por un 0 o bien `DX` (los dígitos hexadecimales se pueden representar de `a–f` o bien `A-F.`

`DXD DX12F OXFFED  DXFFFF 0X12DEFF`

Valores octales de tipo `int` están precedidos por un 0 a la izquierda seguido por dígitos en el rango 0-7

00, 0123, 0777, -045321, -05

Literales de coma flotante

Los números básicos de una coma flotante incluyen un punto decimal.

`1.2345   1234.5678   0.1   305   -24.123`

Si el número de coma flotante termina con un sufijo `f o F` será de tipo `Float.`

`1.23f   3.456F   of   .5Ff   -45673F`

El sufijo `d` o `D` se pueden utilizar para representar tipos `double.`

Regla
Los literales en coma flotante en Java son por defecto `double` precisión .Para especificar un literal `float` se debe añadir una constante `F o f`. Se puede especificar explícitamente un literal de tipo `double` añadiendo `D` o `d.`

Literales boolean
Existen dos valores lógicos que pueden tener un valor *lógico o boolean* `true` (*verdadero)* y `false` (*falso)*

Literales carácter
Un Literal carácter representa un solo carácter encerrado entre comillas simples.

```
´a´, ´A´ ,´¡´,´4´,
```

Tabla A.2 Secuencia de escape

Secuencia de escape	*Descripción*
`\ ddd`	Carácter octal (dddd)
`\ uxxx`	Carácter hexadecimal UNICODE (xxxx)
`\ ´`	Comilla simple
`\ ´´`	Comillas doble
`\\`	Diagonal
`\ r`	Retorno de carro
`\ n`	Nueva línea
`\ f`	Avance de página
`\ t`	Tabulación
`\ b`	Retroceso

Literales de cadena

Los literales de cadena se especifican encerrado una secuencia de caracteres esntre un par de comillas dobles.

```
“Hola Carchelejo”
“cinco/nlineas”
“Esto es una cadena”,”Hola mundo/n”
```

A.6 VARIABLES

Las variables son unidades básicas de almacenamiento en Java .Una variable se define por la combinación de un identificador, un tipo y un inicializador opcional. Además las variables tiene un ámbito que define su visibilidad y una duración.

A.6.1 Declaración de variables

Una variable debe ser declarada antes de poder ser utilizada.

Sintaxis *nombretipo identificador*
no*mbretipo identificador =expresión*

Es posible declarar dos o más variables a la vez.

```
Nombretipo id1, id2,...;
```

Existen dos categorías de variables:

1. *Variables de tipo primitivo* que contienen directamente una representación de un valor de un tipo primitivo.
2. V*ariables de tipo referencia* que contienen una referencia a un objeto cuando un tipo de referencia toma el valor `null`, significa que no se refiere a un objeto real(referencia nula)

A.6.2 Modificaciones de variables
Las variables se pueden modificar mediante indicadores.

`Public`	La clase o variable de instancia es accesible desde todos los ámbitos.
`Protected`	La clase o variable de instancia es accesible solo en el ámbito actual de la clase ,el ámbito del paquete actual y todas las subclases de la clase actual.
`Private`	La clase o variable de instancia es accesible solo en el ámbito actual de la clase.
`Final`	La variable es una constante ,de modo que su valor no se puede modificar.
`Static`	La variable es una variable de clase, compartida entre todos los objetos instancia de una clase.
`Transfert`	Se declara que no es parte de un estado persistente del objeto .
`Volatile`	Se necesita a veces cuando se utiliza una variable instancia por ¿¿ para prevenir al compilador de su optimización.

A.6.3 Inicialización de variables
Se utilizan los siguientes valores por defecto

```
byte              (byte ) 0
short             (short) 0
Int                0
long               0l
float              0.0f
double             0.0d
char               ´/ u000´(carácter nulo)
boolean            false
tipos referncia    null
```

La *inicialización* se realiza de tres formas

1.
```
int i=100;
char c=´d´;
float f=45.325
```

2.
```
int i;
{i=100;}
```

3. asignación dentro del cuerpo de un constructor
```
int a=10,b,c=5;
```

A.6.4 Variables parámetro
Estas variables se inicializan siempre a una copia del valor utilizado en la llamada del método o constructor.

A.6.5 Variables locales
Todas las variables locales deben ser explícitamente directa o indirectamente antes de un uno.

```
{
   int i=10;
      ...
}
```

o por una asignación hecha a la variable antes de ser utilizada en cualquier otra expresión.

```
{   int i;
    // no puede haber ninguna ¿? Que utilice i

        i=?¿
}
```

A.6.6 Variables finales
Las variables que se declaran `final` deben ser inicializadas cuando son declaradas y no pueden ser modificadas.

```
Static final int MAX-CUENTA=100,
```

A.6.7 Conversión de tipos y moldeado
Cuando un tipo de dato se asigna a otro tipo de variable tiene lugar una *conversión automática de tipos* si se cumplen las dos condiciones siguientes:

- Los dos tipos son compatibles
- El tipo destino es más grande que el tipo fuerte.

```
doble a = 6.0
```

```
float b;
b= (float) a;
```

la variable `double` a se convierte a `float` y se asigna a la variable `b`

- Las variables primitivas nunca se pueden moldear a un objeto clase
  `String obj-str=(string)0,  //err`
- Para convertir un entero a una cadena ,añadir un literal de cadena vacía.
  `String obj-str=0+¨¨; // el operador + significa`

CARACTERES ESPECIALES

Los caracteres especiales de difícil representación

Tabla A.3 Caracteres especiales

Sintaxis	**Significado**
`\´`	Comillas simples
`\´´`	Dobles comillas
`\\`	Diagonal
`\b`	Retroceso
`\f`	Avance de página
`\n`	Nueva línea
`\r`	Retorno de carro
`\t`	Tabulación
`\dee`	Representación octal
`\xdd`	Representación hexadecimal
`\udddd`	Carácter *unicode*

A.7 ARRAYS

Un array es un grupo de variables de tipos similares que se conocen con un nombre común. Los diferentes elementos contenidos en un array se definen por un índice y se acceden a ellos utilizando su índice; los índices arrancan en 0. Los arrays pueden ser multidimensionales y al igual que los objetos de la clase se crean utilizando la palabra reservada `new.`

A.7.1 Arrays de una dimensión

Para crear un array se debe crear primero una variable array de tipo deseado. La declaración es

Tipo nombre _van[ ];

```
Int dia_mes [ ];0
```

El operador `new` sirve para asignar memoria y crear un array de una dimensión.

```
Var_array=new tipo[longitud];
Dia_mes=new int [12];  //array de 12 enteros
Dia_mes[1]=28,  //array 28 a dia_mes
```

Sintaxis

```
Tipo variable nombreArray[ ]=new tipo Variable[logArray];
Tipo Variable[ ] nombreArray=new tipovariable[longArray];
```

1. `int [ ] datos =new int [15];`   `int datos [ ];`
`datos=new int[15];`

A.7.2 Arrays multidimensionales

En Java , los arrays multidimensionales son realmente arrays de arrays.

```
Tipovariable [ ] nombrearray=[val1,val2,...valn];
```

Esta sentencia crea un array de `n`. elementos y lo inicializa conlos valores contenidos entre llaves.

La sintaxis completa es:

```
Tipovariable[ ] nombreArray=new tipovariable[ ]{val1,val2,...valn}
```

Ejemplo

```
Int dosD[ ] [ ] new int[4][5];
```

Esta sentencia asigna un array de enteros ,4 por 5 y lo asigna a `dosD`

Sintaxis alternativa de declaración de arrays

Tipo[ ] nombre-var

Las siguientes declaraciones son equivalentes

```
Int a1[ ]=new[int3];
Int [ ]a20newint[3];
```

Las siguientes dos declaraciones son también equivalentes

```
Char mod1[][]=new chat¡r[3][4],
Char [][]mod2=new char[3][4]
```

Acceso y asignación de valores a elementos de un array

La sintaxis para acceder a un elemento es

```
NombreArray[índice]
```

Y para asignar valor al miembro del array se especifica a el `indice`. Por ejemplo:

```
NombreArray[indice]=valor;
```

Por ejemplo

```
Datos[0]=45;
```

A.7.3Arrays de dos dimensiones

TipoVariable []{}nombreArray=new tipoVAriable[filas][columnas].

Crea un objeto array de dos dimensiones. Si el número de columnas se deja en blanco, el array puede tener un número de columnas diferentes por cada final.

Ejemplo

Crear un array bidimensional con 8 filas y un número variable de columnas por cada fila. La primera fila se crea con 20 columnas y la tercera con 100 columnas.

```
class TestArraybidimensional {
  Public static void main (string arrays[]){
    Int[][]multD=new int[8];
    MultD[0]=new int[20];
    MultD[1]=new int[1000];
  }
}
```

Inicialización de un array de dos dimensiones

TipoVariable[]nombreArray={{val1,val2,...},{val1,val2,...}}

Crea un array bidimensional e inicializa las columnas con los valores contenidos entre llaves. Cada conjunto de llaves representa una fila del array bidimensional.

Ejemplo
Inicializar un array bidimensional de valores enteros de modo que se visualice el elemento `[0][2]`

```
  Public class tesArray2D
    Public static void main(string arrays[]){int[][]multiD=}{1,2,3,4,5}{6,7,8,}
    };
    system.out.printl/”El elemento[0][2]es + multiD[0][2];
```

A.7.4 La variable `lenght`

Todos los arrays unidimensionales tienen una variable de instancia denominada `lenght` asociada con ellos. Esta variable contiene la longitud del array. En el caso de arrays bidimensionales, La variable `lenght` se utiliza para acceder a la longitud de la primera fila.

```
  Public class teslongitudinal
      public static void main(sting arrays[]){int[][]multiD=}{1,2,3,4,}{5,6,7}
    };
      system.out.printl (“ la longitud de la 1ªfase +multiD[0].lenght);}
    }
```

Salida

```
La longitud de la primera fila es 4
```

A.8 EXPRESIONES

Las expresiones se utilizan para buscar, calcular y asignar valores. Excepto para una llamada a un método con un tipo de retorno `void,` todas las expresiones devuelvan un valor, permitiendo a las expresiones combinarse en más expresiones complejas
Las expresiones principales traen o crean valores y son las siguientes:

- Las palabras reservadas `this, super y null`
- Un valor literal
- Una expresión con paréntesis
- Una expresión de campo, utilizando ´.´
- Una expresión de índices de array, utilizando ´[]´
- Una expresión de llamada a métodos
- Una expresión de asignación.

Expresión con paréntesis

(expresión)

Expresión de campo

Identificador

```
Expresión principal. Identificador
Paquete. Identificador
```

Expresión de índices de arrays

```
        Término[expresionValorentero]
```

Expresión de asignación

```
        New numbetipo (listaArgumentos)
        New numbetipo [expresionEntera]
```

A.9 OPERADORES

Los operadores permiten a las expresiones combinarse en expresiones más complejas. Java proporciona una colección grande de operadores que se pueden utilizar para manipular datos, incluyendo operadores aritméticos, asignación ,lógicos y de moldeado. Las reglas de *asociación* y de *prioridad* se utilizan para determinar como evaluar expresiones utilizando operadores.

A.9.1 Operadores aritméticos

Los operadores aritméticos se utilizan en expresiones matemáticas de igual modo que se utilizan en Álgebra.

Tabla A.4 Operadores aritméticos

Operador	Significado
+	Suma
-	Resta(también menos unitario)
*	Multiplicación
/	División
%	Módulo
++	Incremento en 1
--	Decremento en 1

Los operadores de incremento y decremento pueden aparecer en formato prefijo `(++variable)` o postfijo(`variable --`). En formato prefijo ,la variable se incrementa o decrementa antes de que se ejecute cualquier operación. Este formato postfijo, la variable se incrementa o decrementa después que se ha ejecutado otras operaciones.

A.9.2 Operadores de asignación

El operador de asignación simple se utiliza para asignar un valor a una variable, vas = expresión. Otros operadores de asignación combinan la asignación con una operación aritmética.

```
Var=var op expresión,   equivale a  var op =expresión;
X + y equivale a x=x+y
```

Ejemplo

```
Int x,y,z;
X=y=z=100;//x,y,z, se ponen a 100
```

Tabla A.5 Operadores de asignación

Operador	Significado
=	Asignación simple
+=	Asignación y suma
-=	Asignación y resta
*=	Asignación y multiplicación
/=	Asignación y división
%=	Asignación y módulo, devuelve el resto del valor de la expresión de la izquierda dividida por el valor de al expresión de la derecha.

A.9.2 Operadores lógicos (bolean)

Los operadores lógicos (bolean) se utilizan para manipular valores `bolean.`

Tabla A.6 Operadores lógicos

Operador	Significado
	&
	AND Lógica
	OR Lógica

XOR lógica(OR exclusiva)
OR cortocircuito(condicional)
ANDcortocircuito(condicional)
NOT unitario lógico
signación AND
signación OR
signación XOR
Igual a
No igual a
Ternario if-then-che.(si-entonces-sino) ¿¿*

Tabla A.7 Tabla de verdad

A	**B**	**AIB**	**A&B**	**A^B**	**!A**
Falso	*falso*	*falso*	*falso*	*falso*	*verdadero*
Verdadero	*falso*	*verdadero*	*falso*	*verdadero*	*falso*
Falso	*verdadero*	*verdadero*	*falso*	*verdadero*	*verdadero*
Verdadero	*verdadero*	*verdadero*	*verdadero*	*falso*	*falso*

A.9.3 Operador ternario (condicional)

Java incluye un operador especial *ternario* que puede reemplazar a ciertos tipos de sentencias `if-then-else.` Su formato es :

`Expresión1 ?expresión2= expresión3`

Expresión1, es cualquier expresión que se evalúa a un valor lógico(boolean). Si *expresión1* es verdadera entonces se evalúa la *expresión2;*El resultado de la operación ? es el de la expresión evaluada. Tanto *expresión2* como *expresión3* han de devolver el mismo tipo de retorno que no puede ser `void.`

```
1 k=i<0 ? –i=i;   se obtiene el valor absoluto de i
2 int i=j<0?5:10, asigna a i si j es menor que 0,y 10 en
                  caso contrario.
```

A.9.4Operadores relacionales

Los operadores relacionales determinan la relación que un operador tiene con otro. Específicamente determina igualdad y ¿¿

Tabla A.8 Operadores relacionales

Operador	Significado
`= =`	Igual a
`! =>`	No igual a
`>`	Mayor que
`<`	Menor que
`>=`	Mayor o igual que
`<=`	Menor o igual que

A.9.6 REGLAS DE PRECEDENCIA

La tabla A.5 muestra el orden de precendecia (prioridad)de mayor o menor. Los paréntesis se usan para alterar la procedencia de la operación.

Tabla A.9 Precedencia de los operadores Java

Mas alta			
()	[ ]	.	
++	--		
*	/	%	
+-			
>>	>>>	<<	
>	>=	<	<=
= =	I=		
&			
^			
I			
&&			
II			
?:			
=	op=		
Más baja			

Los paréntesis elevan la precedencia de las operaciones que están en su interior. Esta técnica se utiliza para obtener el resultado deseado.

A.10 Impresión básica

El flujo de salida estándar permite acceder a los métodos básicos de impresión; `print()` y `println()`,de la clase `PrintStream.` Proporciona una referencia a un objeto de `printStream` que puede acceder a esos métodos .La variable `out` es un miembro de la clase `System.`

Sintaxis System.out // uno de la variable out

A.10.1 Método print( )

`Print( )` imprime el argumento parado a la salida estándar (normalmente la consola). Sin un carácter fin de línea.

Sintaxis

```
public void print(String s)
public void print(Boolean b)
public void print(Char c)
public void print(double d)
public void print(float f)
public void print(int i)
public void print(long l)
public void print(object ob)
```

A.10.2 Método println( )

`Println( )` es similar al método `print( )` excepto un carácter fin de línea o secuencia se añade al final.

```
public void print(String s)
public void print(Boolean b)
public void print(Char c)
public void print(double d)
public void print(float f)
public void print(int i)
public void print(long l)
public void print(object ob)
```

Ejemplo

```
Public clacs PruebaImpresion 1{
  Public static Void main(String args[])}
    For (int i=1;i<5,++i){
      System.out.print(“ línea número”+i);
    }
  }
}
```

=________________________

```
public clacs pruebaImprsion2{
  public static void main(String args[]){
    for (int i=1;i<5;++i){
      system.out.println(”línea número”+i);
    }
  }
}
```

A.11 SENTENCIAS

Sentencias de declaración

tipo nombreVariable;

Ejemplos

```
int longitud;
double e;
Circulo circulo;
```

Sentencias de asignación

Una sentencia se asignación asigna el valor de la expresión en el lado derecho a la variable del lado izquierdo.

```
nombre = expresiónLegal;
```

Ejemplos

```
longitud = 5 + 7;
i += 5;
```

Sentencias `return`

Las sentencias `return` proporcionan una salida de un método con un valor de retorno no `void`. Las sentencias de retorno pueden no aparecer en un método con un tipo de retorno `void`. Las sentencias `return` pueden aparecer en cualquier parte de una estructura de control;producen un retorno inmediato del método. El valor de la expresión a continuación del retorno debe coincidir con el tipo de retorno del método.

Ejemplo

```
public int calcularResta(int x, int y) {
  return x-y;
}
```

Sentencias compuestas

Las sentencias compuestas se encierran entre llaves `{}` y se ejecutan secuencialmente dentro del bloque.

Ejemplo

```
{
  int m = 25;          // asigna el valor 25 a m
  int n = 30;          // asigna el valor 30 a n
  int p = m + n;       // asigna el valor 55 (m + n) a p
}
```

Sentencia `if`

Las sentencias de selección proporcionan control sobre dos alternativas basadas en el valor lógico de una expresión.

```
if (expresiónLógica)
  bloqueSentencias1     //si son varias sentencias se encierran entre
{}
[else if (expresiónLógica)
  bloqueSentencias2]
...
[else
  bloqueSentenciasN]
```

Ejemplo

```
if (i < 0)
  System.out.println("Número negativo");
else
{
  System.out.print("Número válido, ");
  System.out.println("es positivo");
}
```

Sentencia `switch`

La sentencia `switch` es la bifurcación múltiple

```
switch (expresion_int)
{
 case constante_exp1:
   sentencias1;
   /*si se trata de múltiples acciones no es necesario encerrarlas
     entre llaves */
   [break;]
 [case constante_exp2:
   sentencias2;
   [break;]]
 ...
 [case constante_expN:
   sentenciasN;
   [break;]]
 [default
   sentenciasX;
   [break;]]
}
```

Ejemplos

```
1. switch (y / 50)
   {
     case 2: elemento = new Demo2(0, 0); break;
     case 3: elemento = new Demo3(0, 0, 100); break;
     case 4: elemento = new Demo4(0, 0, 200); break;
     case 5: elemento = new Demo5(0, 0); break;
   }

2. switch (n)
   {
     case 1:
     case 2:
```

```
        visualizarResultado("1, 2, Sierra de Cazorla");
        break;
      case 3:
      case 4:
        visualizarResultado("3, 4, Sierra Magina");
      case 5:
      case 6:
        visualizarResultado("3, 6, Sierra de Jaen");
        break;
      default:
        visualizarResultado(n + " fuera de rango");
    } //fin de switch
```

Etiquetas

```
nombreEtiqueta:

break [nombreEtiqueta];
continue [nombreEtiqueta];
```

Ejemplo

```
salir:
      {
        for (i = 0; i < 10; i++)
        {
           for (j = 0; j < 20; j++)
           {
             if (i == 1) break salir;
             System.out.print(j + " ");
           }
            System.out.println();
        }
       } //fin del bloque con la etiqueta
```

Sentencia `while`

La sentencia `while` se utiliza para crear repeticiones de sentencias en el flujo del programa.

```
while (expresiónLógica)
  bloqueSentencias  //el  bloqueSentencias  puede  ejecutarse  de  0  a  n
veces
```

Ejemplo

```
while (cuenta <= numero)
```

```
{
  System.out.print(cuenta + ", ");
  cuenta++;
}
```

Sentencia `do-while`

La sentencia `do-while` se utiliza para repetir la ejecución de sentencias y se ejecuta al menos una vez.

```
do
  bloqueSentencias //el bloqueSentencias se ejecuta al menos una vez
while (expresiónLógica);
```

Ejemplo

```
do
{
  System.out.print(cuenta + ", ");
  cuenta++;
}
while (cuenta <= numero)
```

Sentencia `for`

La sentencia `for` se usa para repetir un número fijo de veces la ejecución de una serie de sentencias

```
for ([iniciación]; [condiciónDeTest]; [actualización])
  sentencias
```

Ejemplo

```
for (int i = 0; i < 10; i++)
  a[i] = 5 * i;
```

Método `exit` y sentencia `break`

La sentencia `break` se puede utilizar en una sentencia `switch` o en cualquier tipo de sentencia de bucles. Cuando se ejecuta `break` el bucle que lo contiene o la sentencia `switch` terminan y el resto del cuerpo del bucle no se ejecuta. Una invocación al método `exit` termina una aplicación. El formato normal de una invocación al método `exit` es

```
System.exit(0);
```

Las estructuras de control proporcionan un medio para controlar el flujo de la ejecución de un programa. Las sentencias de control de un programa en Java se dividen en tres categorías: selección ,iteración y salto. Las **sentencias de selección** permiten a un programa elegir caminos diferentes de ejecución basados en el valor de una expresión o el estado de una variable. Las *sentencias de iteración* permiten la repetición de una o más sentencias (estas repeticiones o iteraciones se conocen como bucles)

Las sentencias de salto o bifurcación permiten a un programa ejecutarse de un modo no lineal.

Sentencias de selección :	`if,switch`
Sentencias de repetición (bucles:)	`while,do,for`
Sentencias de transferencia:	`break,continue, return`

A.11.1Sentencia de selección

La sentencia *if* es una sentencia de bifurcación condicional dependiendo del valor de una expresión lógica.

Sintaxis

```
1 if(expresión lógica)
   sentencia
```

```
2.if (expresión lógica)
   sentencia
     else
       sentencia
```

```
3.if (condición1){
    sentencia;
  }else if (condición2){
    sentencia;
  {else{
    sentencia;
  {
```

```
Ejemplo
```

```
1 if(a= = b){
  c=10;
  d=25;
  }
  h=1.5;
```

```
2. if(a= =b)
   c=10;
   else
   d=25;
   h=1.5;
```

```
3.if (a= =b)
  if(c= =d){
  system.out.printl("c= =d");
  }
  else{
  ¨system.out.printl("c!=d);
  }
```

```
4.public class prueba If{
```

```
  public static void main(String args[]){
  int m=2;
  if(m=  =2){
  system.out.println("m es 2");
  {else {
  system.out.println("m no es 2);
      }
    }
  }
```

Sentencias if-else if en escalerai

```
if (condición)
  sentencia
else if (condición)
  sentencia
else if(condición)
    sentencia
-
-
-
else
  Sentencia
```

Sentencia Switch

La sentencia `switch` es una sentencia de bifurcación múltiple. Proporciona un medio de seleccionar la ejecución de diferentes partes del código dependiendo del valor de una expresión.

```
switch (expresión){
Case valor 1
  //secuencia de sentencias
  break
Case valor 2
  //secuencia de sentencias
  break;
-
-
-
case valor
//secuencia de sentencias
break
default:
//secuencia de sentencias
}
```

A.9.5 Operador

El operador + es un operador de concatenación de cadena `String` cuando uno de los dos operadores, es una cadena (`String)` , en caso contrario se representa la suma aritmética.

Ejemplo

```
public class PuebaCadena {
public Static void main(String args[]){
system .out println(5+6+7+"test");
system. Out. Println("prueba "+5+6+7);
  }
}
```

Salida

```
18 test
Prueba 567
```

Expresión	debe ser de tipo byte,short,int o char
Valor	debe ser un lateral único (una constante, una variable),no se puede duplicar los valores de case.

Ejemplo

```
public class PruebaCase {
  public static void mian(String args[]){
    char letra =´b´;
    int puntos;
    switch(letra){
case´A´;
      Puntos=100;
      break;
case´B´
      Puntos=70;
      Break;
default
      Puntos=0;
}
system.out.println ("puntos="+puntos);
  }
}
```

Sentencias switch anidadas

Se puede utilizar una sentencia switch como parte de la secuencia de sentencias de una sentencia swith externa.

```
switch(contador){
  case 1=
    switch (total){
      case 0=
        system. Out.println("total es cero");
        break;
      case 1:
        system.out.println("total es ceero);
        break
    }
    break;
  case2:...
```

A.11.2 Sentencias de iteración

Las sentencias de iteración o repetición crean *bucles*(lazos). Un bucle se ejecuta repetidamente el conjunto de instrucciones que contiene hasta que se cumpla una condición de terminación.

`While`

El bucle `while` permite que una sentencia o bloque de sentencias se ejecute repetidamente mientras es verdadera una condición o expresión lógica.

Sintaxis

```
while(condición){
// cuerpo de bucle ...sentencia /sentencias
}
```

La *condición* de bucle se evalúa antes de ejecutar el bucle ,de modo que si la condición se evalúa inicialmente.

Ejemplo

```
1 // cuenta de 0 a 9
  int x=0;
  while(x<10){
  system.out.println(x);
  x++;
  }
  //*=10 después de terminar el bucle

2 // prueba del bucle while
  class while{
  public static void main(String args[]){
  int n=10;
  while(n>0){
  system.out.println("cuenta"+n);
  n--;
      }
    }
  }
```

do- while

El bucle `do` permite que una secuencia de sentencias se ejecute repetidamente mientras que una condición lógica se evalúa a verdadera , y se termina en el caso contrario. En una sentencia `do-while,` el cuerpo del bucle se ejecuta al menos una vez.

Sintaxis

```
do
  //sentencias, cuerpo del bucle
while (condición);
```

Ejemplo

```
1 int x=0;
  do
     system.out.println(x);
     x++
   }while(x<10);
   //x=10 después de que termina el bucle

2 // prueba del bucle do-while
   class DoEhile{
   public static void main(string args []){
   int n=10;
   do  {
       system.out.println("cuenta"+n);
       n--;
     }while(n>0);
     }
  }
```

for

El bucle for ejecuta las sentencias del cuerpo un número fijo de repeticiones (interacciones) que depende de una variable de control.

Sintaxis

```
for (inicialización, condición, iteración){
  //cuerpo del bucle
}
```

inicialización se utiliza para inicializar y también ,en el caso de clavar la variable del bucle

```
for(int n=1;...)
```

condición expresión lógica, se evalúa, si es verdadera se ejecuta el cuerpo del bucle si es falsa se termina el bucle.

```
for (...;x<10;...)
```

Iteración es normalmente una expresión que se evalúa después de cada ejecución del cuerpo bucle. Normalmente actualiza (incrementa /decrementa)la variable de control del bucle

```
for(...;...;n++)
```

Ejemplo

```
1 for (int x=1;x<=10;x++)
        //cuerpo del bucle
  }

2  //prueba bucle for
```

```
  class PruebaFor {
  public static void mian(sting args[]){
  int n;
  for (n=10;n>0;n--)
  system.out.println(“cuenta”+n);
    }
}
```

Declaración de la variable de control dentro del bucle.

Si la variable de control del bucle solo se utiliza para la ejecución del mismo, se suele declarar en la sección de inicialización del bucle for.

```
3 //declaración de la variable de control en el bucle
  class Prueba For {
  public static void main(String args[]){
  for(int n=10; n>0;n--)
  system.out.println(“cuenta”+n);
    }
  }
```

Uso de separador de coma

A veces se desea incluir más de una sentencia en las partes inicialización e iteración del bucle `for`. En este caso se separan por el separador coma.

```
4 //uso de la coma
  class coma{
  public static void main(string args[]){
  int a,b;
  for(a=1,b=5; a<b; a++,b--){
  system.out.println(“a= ”)+a);
  system.out.println(“b= ”)+b);
      }
    }
  }
```

Sintaxis :

```
  break;
        break etiqueta;
```

```
1 // break para salir de un bucle
  class BucleBreak {
    public static void main(String args[]){
      for(int 1=0; 1<100;1++)
        if(i= =10)break;//termina el bucle si i es 10
        System. Oout.println(“1:”+i);
      }
      System.out.println(“i:”+i);
    }
```

```
}
```

2. **`break`** como una sentencia `goto` *(transferencia incondicional)*

Sintaxis break *etiqueta;*

```
// break actuado como goto
class Break{
  public static void main(String args[]){
    bolean t=true;
    primero:{
      segundo:{
        tercero:{
          system.out.println("Antes de break");
          If(t)break segundo;
          system.out.println("no se ejecuta");
          }
          system.out.println("no se ejecuta");
        }
      }
    }
  }
```

Bucles anidadas
Java permite anidar bucles

```
  //Bucles anidadas
  class Anidadas {
    public static void main(String args[]){
      int i;j:
        for(i=0,i<10;1++){
          for(j=i;j<10;j++)
            system.out.print(".");
          system.out.println();
      }
    }
  }
```

A.11.3 Sentencias de transferencia
Java soporta tres sentencias de salto o transferencia: `break,continue` y `return.` Estas sentencias transfieren el control a otras partes de su programa.

Break
En Java, la sentencia `break` tiene tres usos:

Primero, termina una secuencia de sentencias en `Switch.`
Segundo, se puede utilizar para salir de un bucle.
Tercero, se puede utilizar como una forma algo más disciplinada que `"goto".`

Continue

`Continue` vuelve el control al principio del bucle

Sintaxis
```
continue;
continue etiqueta;
```

```
// demostración de continue
class Continue{
  public static void main(string args[]){
    for(i=0,i<10;1++){
      system.out.println(i+” “),
      if(i%20 00)continue;
      system.out.println(” “),
    }
  }
}
```

return

La sentencia `return` se utiliza explícitamente para volver desde un método ija que cuando se ejecuta termina inmediatamente el método en que se ejecuta.

Sintaxis
```
return;
return   expresión;
```

```
1 //demostración de return
    class Return{
      public static void main(string args[]){
      bolean t=true;
      System.out.println(“antes de return);
      If(t)return;
      System. Out.println(“no se ejecuta”);
    }
  }

2  int dosPor (int n){
    return n*2;
  }

3 public class Return1{
    private static int min(final int a ,final int b){
      if (a<b)
        return a;
      else
        return b;
    }
    ...
  }
```

A.12 CLASES

}

Una clase es un colección de miembros dato y métodos que define un objeto específico. Es la construcción fundamental del lenguaje Java. Una clase puede ser definida por el usuario o por uno de los paquetes incorporados a Java y se declara con la palabra reservada `class`. El convenio utilizado para nombrar las clases utilizar una letra mayúscula para la primera letra del nombre de la clase.

Sintaxis

```
Class nombreClase
{
  declaración de variables estáticas
  declaración de variables de instancia
  declaración de constructores
  declaración de métodos
  declaración de métodos estáticos
}
  declaración de métodos
    tipo nombreMétodo (lista de parámetros){
      cuerpo del método
    }
```

Los datos o variables, definidas dentro de una clase se denomina Variables de instancia. El código está contenido dentro de métodos .de modo colectivo ,los métodos y las variables definidas dentro de una clase se denominan miembros de la clase. Las variables definidas dentro de una clase se llaman variables de instancia ya que cada instancia de la clase (es decir, cada objeto de la clase) contiene su propia copia de estas variables. Cada clase define un nuevo tipo de datos.

Ejemplo

```
classs caja{            // caja es un nuevo tipo de dato
  double anchura;
  double altura;
  double profundidad;
}

class caja{
  double anchura, altura. Profundidad
  publi x Caja (double a, double h, double p){
    anchura = a;
    altura =h;
    profundidad =p;
  }
  public double obtenerVolumen (){
    return anchura*altura*profundidad;
  }
}
```

C1. Estructura de programas Java

Un programa Java consta de una colección de archivos o unidades de compilación. Cada archivo puede contener un nombre opcional de paquete, una serie de declaraciones `import` y por último una secuencia de declaraciones de interfaces o clases. Una *unidad de compilación* puede identificar sus paquetes, importar cualquier número de otros paquetes, clases o interfaces y declarar cualquier número de clases e interfaces.

Declaración de importaciones

Una declaración de importación (`import`) nombra un elemento de otro paquete que se utilizará en las declaraciones posteriores de interfaces o clases. Se puede utilizar un asterisco para incluir todos los elementos de un paquete.

```
import nombrePaquete.*;
import nombrePaquete.NombreClase;
import nombrePaquete.NombreInterfaz;
```

Así `import java.io.*;` , indica al compilador que importe cualquier clase del paquete `java.io` proporcionado por Java a medida que se necesite. Es una buena idea incluir esta línea al principio de cualquier archivo `.java` que realice operaciones de entrada/salida. Otros ejemplos:

```
import java.util.Date;
import.java.net.*;
```

Definición de clases

Una definición de una clase consta de una declaración y un cuerpo. El cuerpo contiene campos de datos y declaraciones de métodos. La declaración de una clase consta de palabras reservadas e identificadores: una secuencia opcional (en el modelo sintáctico para indicar que es opcional se encierra entre `[]`) de modificadores, la palabra reservada `class`, el nombre de la clase, un nombre opcional de la clase padre, una secuencia opcional de interfaces y el cuerpo de la clase con sus miembros.

```
[modificadoresDeClase] class Nombre [extends Padre]
                                    [implements Interfaz1 [, Interfaz2
[, ...]]]
{
  //cuerpo de la clase (miembros)
}
```

Los `modificadoresDeClase` pueden ser: `abstract` , `final`, `public`

Una clase abstracta es aquella que tiene uno o más métodos abstractos y de la que el programador no piensa instanciar objetos. Su fin es servir como superclase de la que otras puedan heredar. Las clases que heredan de una clase abstracta deben implementar los métodos abstractos de su superclase o seguirán siendo abstractas. Una clase `final`

no puede ser superclase y todos sus métodos son implícitamente `final`. Una clase pública debe estar en su propio archivo denominado *`Nombre.java.`* Los miembros de una clase pueden ser métodos y variables de instancia (pertenecientes a un tipo base o una clase).

```
// Formato más simple de una definición de clase
class ClaseUno
{
  // campos de datos y declaraciones de métodos
} // ClaseUno
```

```
// Una clase que extiende otra clase
public class ClaseDos extends OtraClase
{
  // campos de datos y declaraciones de métodos
} // ClaseDos
```

```
// Clase compleja
public abstract class MiObjeto extends OtraClase implements InterfazUno, InterfazDos
{
  // campos de datos y declaraciones de métodos
} // MiObjeto
```

Ejemplos

```
1. public class PrimerPrograma
   {
     public static void main(String[] args)
     {
       System.out.println("SierraMágina-Carchelejo");
     }
   }
```

```
2. public abstract class Numero
   {
     ...
   }
```

Declaración de variables

En Java, las variables se pueden declarar: (1) como campos de datos de una clase, (2) como argumentos de un método, o (3) como variables locales dentro de un bloque.

Declaraciones de campos de datos y variables de métodos

Una variable se declara proporcionando su tipo y su identificador. El tipo puede ser uno de los tipos primitivos o puede ser una clase. Las declaraciones de las variables locales y campos de datos pueden incluir la asignación de un valor inicial. Los argumentos obtienen su valor inicial cuando se llama al método.

```
//Ejemplos de declaraciones de variables de método o campos de datos
int z;                              // identificador z es de tipo int
char inicialNombre =’M’;            // inicialNombre es de tipo char
                                    // y de valor inicial ‘M’
String saludo = "Hola Mackoy";      // saludo es de tipo String
                                    // y de valor inicial "Hola Mackoy"
boolean interruptor = false;        // interruptor es de tipo boolean
                                    // y valor inicial false
```

Una declaración de variables de instancia o campos de datos tiene una parte de modificador opcional, un tipo, un nombre de variable y una inicialización opcional.

```
[modificadoresDeVariable] tipo nombre [= valor];
```

Los *modificadoresDeVariable* pueden ser: `public`, `protected`, `static`, `final`

Ejemplos

```
1. public class Figura
   {
     protected Rectangulo posicion;
     protected double dx,dy;
     protected Color color;
     //...
   }

2. class Empleado extends Persona
   {
     protected String nombre = "";
     protected int edad;
     protected Empleado unEmpleado;
     //...
   }
```

Visibilidad de campos de datos

Los campos de datos son accesibles desde cualquier método dentro de la clase. Dependiendo de la visibilidad declarada, otros objetos pueden acceder también a los

campos de datos. A los campos de datos que no se les proporciona un valor inicial explícito se les asigna un valor por defecto.

Declaración de constantes de clase

Las constantes de una clase se declaran como variables, siendo necesario comenzar su declaración con las palabras reservadas `final y static` y se les asigna un valor en la declaración. Este valor ya no se podrá modificar.

Ejemplo

```
class Empleado extends Persona
{
  public static final cantidad = 50;
  //declaración de variables

  //declaraciones de métodos
}
```

Conversión explícita de tipos

(nombre_tipo) expresión

Creación de objetos

Una *instanciación* (creación) de *objetos* crea una instancia de una clase y declara una variable de ese tipo. Los objetos se crean a partir de una clase utilizando el operador `new`. La sintaxis adecuada es:

```
[tipo] nombreVariable = new tipo([parámetro1[, parámetro2[, ...]]]);
```

```
Repuesto unaPieza = new Repuesto();
Automovil miCarro = new Automovil(5, "Golf");
```

La creación de una instancia (un objeto)

Crea un objeto con el nombre nombreVariable
Le asigna memoria dinámicamente
Inicializa sus variables de instancia a los valores por defecto: null para los objetos, false para variables booleanas, 0 para los otros tipos base
Llama al constructor con los parámetros especificados
Por último devuelve una referencia al objeto creado, es decir la dirección de memoria donde se encuentra dicho objeto.

Declaración de métodos

Las declaraciones de métodos simples, denominadas también *signaturas*, constan de un tipo de retorno, un identificador, y una lista de argumentos (parámetros). El tipo de retorno puede ser cualquier tipo válido (incluyendo una clase) o el tipo void si no se devuelve nada. La lista de argumentos consta de declaraciones de tipo (sin valores iniciales) separados por comas. La lista de argumentos puede estar vacía. Los métodos pueden también tener una visibilidad explícita.

```
[modificadoresDeMétodos] tipoDeResultado nombreMétodo
                                ([tipoParámetro1 parámetro1
                                 [,tipoParámetro2 parámetro2[, ...]]])
                              [throws Excepción1[, Excepción2[,...]]]
{
  //cuerpo del método
}
```

Los `modificadoresDeMétodos` pueden ser: `public`, `protected`, `private`, `abstract`, `final`, `static`, `synchronized`. Como `tipoDeResultado` se especificará `void` cuando el método no devuelva resultados. En la implementación del método, cuando éste no haya sido declarado `void`, se utilizará la instrucción `return` para devolver un valor al punto de llamada del método. Es decir que, en cuanto que se ejecuta `return`, el método termina devolviendo un único valor como resultado. Para devolver múltiples valores mediante una función en Java deben combinarse todos los ellos en un objeto y devolver la referencia al objeto. A continuación del nombre del método y entre paréntesis se especificará la lista de parámetros, que constará de cero o más parámetros formales cada uno de ellos precedido por su tipo y separados por comas.

Cuando se llama a un método, los parámetros actuales se asignan a los parámetros formales correspondientes. Entre los parámetros actuales, los de la llamada, y los formales, los de la declaración, debe existir concordancia en cuanto a número, tipo y orden.

La palabra reservada `throws` permite listar tipos de excepciones lanzadas por el método cuyo tratamiento se pospone para que sea efectuado por el método llamador.

Los métodos de una clase están asociados con una instancia específica de la misma, excepto si son estáticos.

```
public class Ejemplo1 {
  // campos de datos declarados, ninguno
  // Declaración simple: no se devuelve nada, no se pasa ningún argumento
  private void calcularImpuestos(){
      // cuerpo del método
  }
  // Un método con un argumento de tipo double que devuelve un entero
  public int calcularTotal (double x){
      // cuerpo del método
  }
```

```
  /* Un método que devuelve un objeto de tipo MiObjeto  con un entero
y
     una cadena de entrada */
  protected MiObjeto convertir(int z, String s) {
     // cuerpo del método
  }
}  // clase Ejemplo1
```

Llamadas de métodos

Cuando se llama a un método, se deben proporcionar los argumentos del tipo adecuado

```
// interior de un método
{
  calcularZ();
  int z = calcularZ(16,25);
  MiObjeto obj = convertir(25, " Hola Mackoy" );
  ...
}
```

El método `main`

Cada aplicación Java (no los *applets*) debe tener un método `main` que es donde comienza la ejecución de la misma. Es decir que, para ejecutar un programa el intérprete de Java comienza llamando al método `main()`. Este método se llama antes de la creación de un objeto y ha de declararse como `static` para que se pueda llamar sin tener que referirse a una instancia particular de la clase. Como además es llamado por código fuera de su clase también tiene que ser declarado como `public`, que es la forma de permitir que un miembro de una clase pueda ser utilizado por código que está fuera de la misma. La palabra reservada `void` indica que `main` no devuelve nada.

```
public static void main(String[] args)
{
}
```

`String[] args` es la declaración de un array de `String`, mediante el cual la clase podría tomar un número variable de parámetros en la línea de órdenes; aunque no se use es necesario incluir este parámetro cuando se define el método `main()`

Extensión de clases

```
[acceso] [final] class NombreClase extends Superclase
{
  // cuerpo de la clase ampliada
}
```

Constructor de la subclase

```
public NombreClase(arg11, ...)
{
  super(...);
  ...
}
```

Constructores

La sintaxis de un constructor es similar a la de un método, sin `tipoDeResultado` y cuyo nombre debe coincidir con el de la clase. El constructor se invoca automáticamente cuando se crea una instancia de la clase.

```
[modificadoresDeConstructor] nombreConstructor
                         ([tipoParámetro1 parámetro1
                          [,tipoParámetro2 parámetro2[, ...]]])
{
  //cuerpo del constructor
}
```

Los `modificadoresDeConstructor` siguen las mismas reglas que en los métodos normales, pero un constructor abstracto estático final no está permitido.

Un constructor debe ser invocado con el operador `new`.

Una clase puede tener múltiples métodos constructores, siempre que éstos se diferencien unos de otros en el número y/o tipo de parámetros.

```
class Persona
{
  protected String nombre = "";
  protected int edad = 0;
  public Persona(String nom, int años)
  {
    nombre = nom;
    edad = años;
  }
  public static void main(String args[])
  {
    Persona p = new Persona("Luisito Mackoy", 13);
    System.out.println("Nombre: "+ p.nombre + "  " + "Edad: " + p.edad);
  }
}
```

Los constructores en la extensión de clases

El *cuerpo de un constructor* comienza con una llamada heredada al constructor de la superclase de la clase. Esta llamada debe ser la primera sentencia del cuerpo de un constructor y no puede aparecer en ningún otro lugar. En Java `super(...)` es usado en

vez del nombre del constructor de la superclase. Si no se usa `super` entonces se supone implícitamente que el cuerpo del constructor comienza con la llamada `super()` sin parámetros. El resto del cuerpo es como un método normal

```
class Empleado extends Persona
{
  protected String categoria = "";
  protected int salario = 0;

  public Empleado(String nom, int años, String nivel, int sueldo)
  {
    super(nom, años);
    categoria = nivel;
    salario = sueldo;
  }
  public static void main(String args[])
  {
    Empleado e = new Empleado("Arturito Mackoy", 13, "medio", 200000);
    System.out.println("Nombre: " + e.nombre + "  " + "Edad: " + e.edad);
    System.out.println("Nivel: " + e.categoria + "  "
                       + "Salario: " + e.salario);
  }
}
```

Definición e implementación de interfaces

Definición de una interfaz

```
public interface NombreInterfaz
{
  public abstract tipoDeResultado nombreMétodo();
  //otras declaraciones de métodos vacíos.
}
```

Se ha de tener en cuenta que:

- Todos los miembros de una interfaz son públicos automáticamente
- Todos los métodos son abstractos automáticamente
- Todos los campos deben ser declarados `static` y `final`

La clase que implementa la interfaz debe implementar todos los métodos declarados en ella

```
public class Implementa [extends Padre] implements NombreInterfaz
{

  public tipoDeResultado nombreMétodo()
  {
    //...
```

```
  }
  //se implementan todos los métodos de la interfaz NombreInterfaz
}
```

Es posible que una clase implemente más de una interfaz

```
[modificadoresDeClase] class Nombre [extends Padre]
                                    [implements Interface1
                                             [, Interface2 [, ...]]]
{
  //Implementación de todos los métodos de las distintas interfaces
}
```

Es posible definir clases que tengan objetos del tipo `NombreInterfaz`, como si la interfaz fuera una clase, pudiendo así usarse en ellas, las diversas implementaciones de ésta. La clase `Ejemplo` puede usar, entre otras que hubiera definidas, la que ofrece la clase `Implementa`.

```
public class Ejemplo
{
  public Ejemplo()
  {
    //...
  }
  public tipoDeResultado unMétodo(NombreInterfaz elemento)
  {
    //...
  }
}
```

Clases anónimas

Una *clase anónima* es aquella que no tiene nombre y, cuando se va a crear un objeto de la misma, en lugar del nombre se coloca directamente la definición.

```
new SuperNombre() { cuerpo clase }
```

Por ejemplo, considerando declarada una clase `Implementa` que implementa `NombreInterfaz`, la siguiente instrucción

```
e.unMétodo(new Implementa());
```

pasa a `e.unMétodo` una nueva instancia de dicha clase `Implementa` como parámetro. Si se quisiera emplear una clase anónima no se efectuaría la declaración de `Implementa` y la instrucción anterior se sustituiría por

```
e.unMetodo(new NombreInterfaz()
```

```
{
   public tipoDeResultado nombreMétodo()
   {
     //...
   }
// SE IMPLEMENTAN TODOS LOS MÉTODOS DE LA INTERFAZ NOMBREINTERFAZ
}
);
```

A.12.1 Privilegios de acceso

El uso de las palabras reservadas `public`,`private` y `protected` con una declaración permite al programador controlar explícitamente el acceso a una variable, método, clase anidada o interfaz anida desde fuera del ámbito de una clase .

Modificaciones de acceso

Las modificaciones de acceso se colocan delante de al declaración de las clases:

```
public, protected, private
```

El uso de un modificador de acceso es opcional y se puede omitir.

`public`	una declaración es accesible por cualquier clase.
`protected`	una declaración es accesible a cualquier subclase de la declaración de la clase o a cualquier clase del mismo paquete.
`private`	una declaración sólo es accesible desde dentro de la clase dDonde está declarada.

A.12.2 Creación de un objeto

Un objeto de una clase se crea mediante la sentencia `new`

Ejemplo: crear un objeto `micaja`

1. `caja micaja= new caja(1;`
2. `caja micaja;micaja= new caja();`
3. `caja c,`
   `c=new Caja (2.0,3.0,4.5);`
4. `Alternativamente`
   `Caja c=new caja (2.0,3.0,4.5);`

A.12.3 Métodos

Un método proporciona la implementación del comportamiento dinámico de los objetos y puede cambiar el estado de los objetos a que llaman.

Sintaxis

```
Modificaciones tipo nombreMétodo (lista de parámetros){
  Secuencia de sentencia
}
```

Los métodos que tienen un tipo de retorno distinto de void devuelven un valor utilizando la sentencia return

Return valor;

```
Valor es el valor devuelto
```

Ejemplo

```
class caja{
  double anchura;
  double altura;
  //calculo y devuelve el volumen
  double volumen(){
    return anchura *altura *profundidad;
  }
}
```

Sintaxis

```
NombreCostructor(lista parámetros){
```
Cuerpo del constructor

Ejemplo

```
1 class caja {
    double anchura;
    double altura;
    double profundidad;
    //constructor
caja (){
system.out.printlh(“ construcción de la caja”);
    anchura=10;
    altura=20;
    profundidad=5;
//calculo y devuelve el volumen
double volumen (){
    return anchura *altura *profundidad;
  }
}
```

Constructores con parámetros
Aunque el sistema anterior es práctico, todas las cajas tienen las mismas dimensiones. Se necesita un medio para construir objetos `caja` de dimensiones diferentes. El método más fácil es añadir parámetros al constructor.

Métodos con parámetros

```
1 int cuadrado ()                  2 int cuadrado (int l)
  {                             {
```

```
  return 12*12;                         return l*l;
  }                             }

3 void fijarDim(double a,double h,double p){
    anchura=a,
    altura=h;
    profundidad=p;
}
```

A.12.4 Constructores

Todos los objetos se deben inicializar cuando se crean, de modo que ellos “nacen”con un estado bien definido. Un *constructor* es un tipo especial de método que inicializa un objeto inmediatamente para su creación. Tiene el mismo nombre que la clase y sintácticamente es similar a un método. Los constructores no devuelven ningún tipo y se llaman automáticamente cuando se crea un objeto de la clase.
El constructor se puede sobrecargar para dar múltiples constructores con diferentes tipos de argumento parado. Los constructores se definen generalmente como `public.`
Los valores separan al constructor situándolos dentro e los paréntesis de las sentencias de creación de objetos de la clase.

```
class caja {
double anchura ,altura, profundidad;
    anchura=a,
    altura=h;
    profundidad=p;
}

//calculo y devuelve el volumen
double volumen (){
    return anchura *altura *profundidad;
  }
}
```

A.12.5 La palabra reservad this

La palabra this representa una referencia que se pasa ,implícitamente a cada método no estática. La referencia es al objeto innovador .

```
1 Caja (double a,double h,double p){
    this. anchura=a,
    this .altura=h;
    this .profundidad=p;
  }

2 class punto
  int x,y;
  punto(int x,int y){
    this.x=x;
    this.y=y;
  }

  int leer x(){
```

```
    return x;
  }

  int leer y(){
    return y;
    }
  }
```

A.12.6 El método `finalize()`

A veces un objeto necesita realizar alguna acción cuando se destruye. Para manejar tales situaciones Java proporciona un mecanismo llamado *finalización.* Para añadir un *finalizador* a una clase, se define simplemente el método finalize()

Sintaxis

```
  protect void finalize()
  }
    //código de finalización
  }
```

A.13 SOBRECARGA DE MÉTODOS

En Java es posible definir dos o más métodos en la misma clase, que compartan el mismo nombre, mientras que las declaraciones de sus parámetros es diferente. En este caso los métodos se dice que están *sobrecargados* y el proceso se conoce como *sobrecarga de métodos.*

La sobrecarga de métodos es una de las formas en Java las cuales implementa el polimorfismo.

Un método de una subclase que tiene el mismo nombre pero diferentes argumentos de un método de una superclase se considera como una función sobrecargada. El compilador decide cual es el método más adecuado basándose en la lista de argumentos y el tipo del objeto que hace la llamada.

Ejemplo

```
  //sobrecarga de método
  class demosobrecarga {
    void prueba (){
      system.out.println(“ningún parámetro”)
    }

  //sobrecarga de prueba con parámetro entero
    void prueba (int a){
      system.out.println(“a:”+a);
}

  //sobrecarga con dos parámetros enteros
    void prueba (int a,int b)
{
      system.out.println(“a y b”+a+” “+b);
}
```

```
class sobrecarga {
  public static void main(string args[]){
  demoSobrecarga ob=new DemoSobrecarga ();
  int resultado;

  //llamada métodos sobrecargados
  ob.prueba();
  ob.prueba(15);
  ob.prueba(5,25);
  }
}
```

Ejemplo2

```
public class Prueba1{
  public static void main(string args[]{
    subclase s=new Subclase();
    system.out.println("el mensaje es" + s.leerMensaje());
  }
}

class Superclase{
  public String leerMensaje(){
    return "Super mensaje";
  }
}

class Subclase extends superclase{
  public String leerMensaje(){
return "Super mensaje";
  }
}
```

Salida

```
El mensaje es submensaje
```

Tipos especiales de clases
Existen tres tipos especiales de clases: abstractas (`abstarct`), finales(`final`), internas (inner).

A.13.1 Clases abstractas
Las clases abstractas se utilizan para definir la estructura general de una familia de clases derivadas. Se utilizan, como los interfaces, para imponer una funcionalidad común a un número de clases deferentes que se conocen como clases concretas. Una *clase concreta* es una clase que no es abstracta. No se puede crear objetos de una clase abstracta, sólo se pude crear de una clase concreta.

> *Las clases abstractas no se pueden instanciar , pero se pueden hacer referencias de un tipo abstracto que se pueden utilizar para apuntar a una instancia de un objeto de una subclase concreta.*

Las clases abstractas pueden definir métodos abstractos y concretos. Cualquier subclase concreta de una clase abstracta debe implementar los métodos abstractos de la clase abstracta. Las clases abstractas se designan utilizando las palabras clave `abstact` delante de la palabra clave `class` en la sentencia de declaración de la clase. Los métodos *abstractos* se declaran con el siguiente formato.

```
  abstract tipo nombre  (lista de parámetros)
```

Ejemplo

La clase abstracta `Figura` define el método `calculararea().`Dos subclases de `figura`, `rectángulo` y `triangulo` proporcionan dos implementaciones deferentes de `calculaearea()`

```
abstract class Figura{
  double dimi dmj;
  figura(double a,double b){
    dimi=a;
    dimi=b;
  }
  abstact double clacularArea();  // método abstracto
}

class Rectángulo extends figura{
  Rectángulo (double a; double b) {
    Super (a,b);
}
double calcularArea(){
  return dimi*dmj;
}
}

class triangulo extends Figura{
  triangulo (double a, double b){
    Super(a,b);
  {
  double calcularArea(){
    return 0.5*dimi*dmj;
  {
{
```

A.13.2 Clases internas

Las *clases internas* permiten que una clase esté contenido en otra clase. Existen cuatro tipos de clase internas:

- Clases miembro estáticas
- Clases miembro
- Clases locales
- Clases anónimas

```
// Muestra de una clase interna
class Externa {
  int externa_x=50;
  void prueba(){
    Interna interna =new Interna();
    Interna. presentan ();
}

//clase interna.
Class Interna{
  void presentan();

  System.out.println("presentan:externa_x="+externa_x);
    }
  }
}

class DemoClaseInterna{
  public Static void main(String args[]){
    Externa  externa= new Externa();
    Externa.prueba();
  }
}
```

A.13.3 Clases finales

Una clase final es aquella que no puede ser heredada. Esta acción se consigue precediendo a la declaración de la clase con `final`. No es legal declarar a una clase como abstracta (`abstract`) y final (`final`).

Ejemplo

```
Final class A{
  //...
}

// La siguiente clase no es legal
class B extends A {  //error,no es posible subclase A
  //...
}
// A   Se declara final y por tanto B no puede heredar de A
```

A.14 HERENCIA

La herencia permite crear una clase conocida como *subclase o clase hija a* partir de otra clase conocida como *superclase o clase padre*. Por consiguiente, una subclase es un versión especializada (una extensión) de una superclase. Se heredan todas las variables de instancia y los métodos definidos en la superclase y se añaden elementos propios.

Sintaxis

```
class nombresubclase extends nombreSuperClase {
  Declaraciones de variables y métodos
```

Solo se puede especificar una superclase par cualquier subclase que se crea. Este tipo de herencia se denomina *herencia simple.* Java no soporta la herencia de múltiples superclases. Es decir no soporta *herencia múltiple* . *Solo puede existir una superclase después de la palabra clave* `extends`.

REGLA

Un objeto de la subclase tiene acceso a los métodos públicos y miembros dato y protegidos y definidos en al superclase; no se puede acceder a los miembros de la superclase que hayan sido declarados parados.

```
class A{
  int i;             // i, es público
  private int j;     //j, es privado, solo se accede en A
  void leerij(int x,int j){
    i=x;
    j=y;
  }
}


class B extends A {
  int totsl,
  void suman(){
    total =I+j;  //error,j no es accesible aquí.
  }
}
```

A.15 PAQUETES

Un *paquete* es un contenedor de clase que se utiliza para dividir o compartimentar el espacio del nombre de la clase. Los paquetes se almacenan de modo jerárquico y se importan explícitamente en nuevas definiciones de clase.

A.15.1 Definir un paquete

Crear un paquete es muy fácil: basta incluir una orden `package` como primera sentencia en el archivo fuente de Java. Cualquier clase declarada dentro de un archivo pertenece al paquete especificado. La sentencia `package` define un espacio de nombres en el que se almacenan las clases. Si se omite la sentencia `package,` los nombres de la clase se ponen en el paquete por defecto que no tiene nombre.

Sintaxis

```
  Package nombrepaquete;
  Package Mipaquete ;// crea un paquete Mipaquete
```

Se puede crear una jerarquía de paquetes

Sintaxis

```
Package paq1[.paq2][.paq3];
Package java.awt.image;
```

Java utiliza los directorios de los sistemas de archivo para almacenar paquetes. por ejemplo, los archivos. `Class` de que se crean como parte de un porque se deben almacenar en un directorio que tenga el nombre que el paquete.

Ejemplo

Los archivos `.class` de las clases del paquete `demo.upsa` se deben almacenar en un directorio denominado `upsa` dentro de un directorio denominado `demo.`

A.15.2 Acceso a paquetes (importación)

A los paquetes y clase de las bibliotecas se accede utilizando la palabra reservada `import`

Sintaxis

```
import nombrePaquete.nombre Clase;
import nombrePaquete;
```

Ejemplo

```
import java.util.dato;    //acceso a la claseDato del
                          //paquete java.util
import jav.io.*;          //se importa todo al paquete.
```

Todas las clases estándar Java incluidas con Java se almacenan en un paquete denominado `Java.` Las funciones básicas del lenguaje se almacenan en un paquete dentro del paquete `Java` llamado `java.lang`

```
Import java.lang.*
```

A.16 INTERFACES

Los interfaces se utilizan para imponer una cierta funcionalidad en las clases que las implementan. Es decir, se utilizan para abstraer el interfaz de una clase de su implementación. Utilizando interface se puede especificar lo que hace una clase pero no como lo hace. Los interfaces son muy similares sintácticamente a las clases pero no tienen implementación de métodos.

Sintaxis

```
TipoACero interface nombre {
  Tipo_retorno nombreMétodo 1(lista de parámetros);
  Tipo_retorno nombreMétodo 2 (lista de parámetros);
  // otros métodos.
  Tipo nombreVar 1= valor;
  Tipo nombreVar 2= valor;
  // otras variables
```

```
}
```

Ejemplo

```
1 interface rellamada {
  void rellamada (int param);
  }

2 interface Area {
  double leerArea();
  }
```

A.16.1 Implementación de interfaces

Una vez que se ha definido una interfaz (`interface`), se puede implementar una o más clase de una interfaz. Para implementar una interfaz, se incluye la clausula `implements` en la definición dela clase y a continuación se crean los métodos definidos por la interfaz

Sintaxis

```
  Acceso class nombreclase [extends superclase]
      [implements interfaz [,interfaz...]]{
      // cuerpo de la clase
  }
```

Ejemplo

```
  // La clase Cliente implementa la interfaz rellamada
class Cliente implements de Rellamada
    // implemento la interfaz de rellamada
    public void rellamada (int p){
        system.out.println("la rellamada con "+p);
    }
  }
```

A.17 EXCEPCIONES

Sentencia `try-catch`

La captura de excepciones se realiza mediante bloques `try-catch`. La/s sentencia/s de un bloque se colocarán siempre entre llaves

```
try
  bloqueAIntentar  //aunque sea una única sentencia ésta irá entre {}
catch (tipoExcepcion1 identificador1)
  boqueSentencias1
[catch (tipoExcepcion2 identificador2)
  boqueSentencias2]
...
```

```
[finally
  boqueSentenciasN]
```

o bien

```
try
  bloqueAIntentar
finally
  boqueSentenciasN
```

ya que el bloque `try` no puede aparecer sólo

Ejemplo

```
import java.io.*;
public class ReturnTryEj
{
  public static int leer()
  {
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String cadena = "";
    try
    {
      cadena = br.readLine();
      return Integer.parseInt(cadena);
    }
    catch(Exception e)
    {
     if (e instanceof IOException)
         System.out.println("Error de entrada/salida");
     else if (e instanceof NumberFormatException)
         System.out.println("No tecleó un número entero");
    }
    // Instrucción siguiente a catch
    System.out.println("Se devuelve 0");
    return 0;
  }

  public static void main (String args[])
  {
    int n;
    do
    {
      System.out.print("Deme un número entero entre 1 y 20 ");
      n = leer();
    }
    while ((n <= 0) || (n > 20));
    System.out.println("2^" + n + " = " + Math.pow(2,n));
```

```
  }
}
```

Sentencia `throw`

Una sentencia `throw` lanza una excepción, que puede ser una excepción recibida o bien una nueva excepción. Una claúsula `catch` puede recibir una excepción y, en lugar de tratarla o después de hacerlo, volver a lanzarla mediante una instrucción `throw`

```
try
bloqueAIntentar
catch(NumberFormatException identificador)
{
  //...
  throw (identificador);
}
```

Para lanzar una nueva excepción se crea un objeto de una subclase de `Exception` que implemente un constructor y se lanza con `throw` cuando ocurra el hecho que debe provocar la excepción

```
if (expresiónLógica)
  throw new ConstuctorSublclaseException([parámetro1[, parámetro2
                                         [, ...]]]);
```

Ejemplo

```
if  (edad < 18 || edad  >  65)
   throw new FueraDeRango("Excepción: valor fuera de rango");


class FueraDeRango extends Exception
{
  String mensaje;

  public FueraDeRango(String causa)
  {
    mensaje = causa;
  }
  public String getMessage()
  {
    return mensaje;
  }
}
```

Sentencia `throws`

Lista las excepciones no tratadas y pertenecientes a clases distintas de `RuntimeException`. Así su tratamiento será pospuesto y deberá ser efectuado por el método llamador o tendrán que volver a ser listadas en la cabecera de éste con otra cláusula `throws`

```
[modificadoresDeMétodos] tipoDeResultado nombreMétodo
                                ([tipoParámetro1 parámetro1
                                 [,tipoParámetro2 parámetro2[, ...]]])
                                 [throws                  Excepción1[,
   Excepción2[,...]]]
{
  //cuerpo del método que no trata la excepción
}
```

Sentencia package

Cada clase pública definida en Java debe ser almacenada en un archivo separado y si hay varias relacionadas todas ellas se almacenan en el mismo subdirectorio. Un conjunto de clases relacionadas definidas en un subdirectorio común puede constituir un paquete Java. Los archivos del paquete deben comenzar con la siguiente sentencia

```
package nombrePaquete;
```

donde el nombre del paquete refleja el subdirectorio que contiene dichas clases. Se utiliza el carácter punto `(.)` como separador entre nombres de directorios, cuando es necesario especificar varios para referenciar al que contiene las clases .

Ejemplo

```
package libro.Tema03;
```

Se puede usar una clase definida en otro paquete especificando, para referirse a ella, la estructura de directorios del otro paquete seguida por el nombre de la clase que se desea usar y empleando el carácter punto como separador. Referenciar clases de esta forma puede resultar molesto y la solución consiste en utilizar `import`, que permite incluir clases externas o paquetes enteros en el archivo actual

C3. Miscelánea

Referencia a miembros de una clase

nombreObjeto.nombreComponente

Si es `static` no es necesario referirse a una instancia en particular de la clase y puede referenciarse como

nombreClase.nombreComponente

Los miembros de una clase están asociados con una instancia específica de la misma, excepto si son estáticos.

Conversión explícita de tipos

Existen dos tipos fundamentales de conversiones de tipos que pueden ser realizados en Java, con respecto a tipos numéricos y con respecto a objetos. El formato a aplicar para efectuar una conversión explícita de tipos es:

`(`*`tipoNombre`*`)` *`expresión`*`;`

Ejemplo

```
double resultado = (double) (4/8);      //asigna 0.0 al resultado
double resultado = (double)4/(double)8; //asigna 0.5 al resultado
double resultado = (double)4/8;         //asigna 0.5 al resultado
double resultado = 4/8; //conversión implícita, asigna 0.0 al
                        // resultado
```

```
Alumno unAlumno = (Alumno)unaPersona;
```

Una *excepción* es una condición anormal que se produce en una sentencia de código a tiempo de ejecución. En otras palabras, una excepción es un error en tiempo de ejecución. Una excepción Java es un objeto que se describe una condición excepcional (esto, es error) que ha ocurrido a un segmento de código. Cuando surge una condición excepcional , se crea un objeto que representa una excepción y se lanza (throw) en el método que produce el error.

El manejo de excepciones se gestiona mediante cinco palabras reservadas: `try, catch, throw, throws y finally`

Sintaxis de bloque de manejo de excepciones

```
try {
  // bloque de código par monitorizar errores
}

catch (tipo Excepción 1 exob){
  // manejador de excepciones para tipoExcepción 1
}

catch (tipo Excepción 2 exob){
  // manejador de excepciones para tipoExcepción 2
}
//...
finally{
  // bloque de código que se ejecuta antes de terminar
bloque try
}
tipoExcepción  tipo de excepción que ha ocurrido
```

Sintaxis bloque `try-catch`

```
try {
```

```
    código
  } catch (excepción e){
    código ;
  {
```

sintaxis *finally*

```
  finally{
    código;
  {
```

Sintaxis *throw*

```
  Throw objetolanzable;
```

Sintaxis *throws*

…

```
nombreMétodo  throws Excepción 1, Excepción 2,...
```

Ejemplo

```
  public class PruebaExcepción {
    public static void main(String
      int [] a =
        0,1,2,3,4,5
      };

      try {
        for (int I=0; I<6; ++I){
          System.out. println(“a[“+I+”]es”+a[I]);
        }
      } catch (excepción e){
          System.out.println(“Excepción :”+e);
      }
      System.out.println(“codigo que se puede ejecutar”);
    }
}
```